Utilizing Scenarios in the Software Development Process

Kevin M. Benner, Martin S. Feather, W. Lewis Johnson and Lorna A. Zorman*

USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, USA
e-mail: {benner,feather,johnson,lorna}@isi.edu

[In N. Prakash, C. Rolland, and B. Pernici, editors, *Information System Development Process*, IFIP Transactions A-30, Elsevier Science Publishers, September 1993. Mr. Benner's current email address is kbenner@andersen.com.]

Abstract

Scenarios play an important role throughout the information system development process. Scenarios are partial descriptions of system and environment *behavior* arising in restricted *situations*. They are instrumental to the following activities: describing and clarifying the relevant properties of the application domain, uncovering system requirements, evaluating design alternatives, and validating designs. This paper will describe these roles in the context of an example and explain how computer-based tools can support the use of scenarios throughout the development process. The thesis of this paper is based on experience with three such computer-based tools.

Keycodes: D.2.1; D.2.6
Keywords: Requirements/Specification; Programming Environments

1 Introduction

This paper argues that scenarios have an important role to play in the information system development process. Scenarios are partial descriptions of system and environment behavior arising in restricted situations. Both the behaviors and the situations are expressed in concrete terms. Using scenarios in information system development usually involves an interplay between describing situations and describing behaviors; e.g., an analyst might propose a situation and decide what behavior would be appropriate to that situation, or suggest a behavior and determine in what situations that behavior might arise.

*The authors have been supported in part by Rome Laboratory contract F30602-89-C-0103 and in part by Defense Advanced Research Projects Agency contract DABT 63-91-K-0006. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion or policy of RL, DARPA, the U.S. Government, or any other person or agency connected with them.

Conventional specification and implementation languages are not designed to support the description of systems via scenarios. Specifications describe the requirements that the system must satisfy in all cases. Likewise, source codes are sets of instructions sufficient to handle all possible inputs that the system will encounter. This emphasis on general, all-encompassing descriptions in software engineering stands in stark contrast to most human expertise. Cognitive science has demonstrated that human expert knowledge is organized in chunks, which recognize and respond to specific situations [1]. Human problem solving tends to be highly situated, i.e., people respond to details of the situation as it unfolds rather than carry out detailed plans that have been worked out in advance [19]. This incompatibility between software engineering notation and human expertise causes severe difficulties in requirements elicitation and system validation. System analysts must elicit from domain experts general properties of the application domain, and general rules of behavior that the system should follow. Experts may be able to offer such rules, but when they are presented with particular hypothetical situations the experts are likely to raise issues that are not accounted for in the general rules. Likewise, validating the implemented system against client intent involves evaluating the system's behavior in specific situations.

Scenarios have attracted interest lately in the human-computer interaction (HCI) community, as a way of describing sequences of interactions between systems and users [5, 16]. Campbell notes [4] that they can be used to illustrate how a user might accomplish particular tasks with the system, to evaluate system usability, to guide interface design, and to test theories of human-computer interaction. There has been limited discussion in the software engineering community of their use for describing, critiquing and explaining system behavior [2, 8]. Empirical studies have shown that designers evaluate designs by mentally simulating scenarios [9].

We claim that the significance of scenarios is greater than even these studies suggest. Scenarios are pervasive throughout the software development process, and are important for any system that is situated in a complex environment, not just systems with human-computer interfaces [12]. Furthermore, computer-based tools can enable scenarios to be realized as explicit artifacts. Automated retrieval, execution, etc. of scenarios can supplement and support the activities that software engineers perform mentally on scenarios at the present time.

In what follows, four different uses for scenarios will be discussed:

- describing and clarifying the relevant properties of the application domain,
- uncovering system requirements,
- evaluating design alternatives, and
- validating designs.

The rest of the paper is structured as follows: Section 2 introduces the example that we will use as illustration, the design of a traffic-light control system for a 4-way intersection. Section 3 elaborates our definition of a scenario. Section 4 introduces some scenario

processing capabilities that we are developing. Section 5 illustrates our points regarding the different uses for scenarios by appealing to the 4-way intersection example. Section 6 enumerates the kinds of automated support that computer-based tools might be able to provide for scenarios in these activities, and describes and assesses the tools that we have developed so far for scenario processing. Section 7 summarizes and concludes the paper.

2 An Example Application

The following problem will be used throughout this paper as a focus for discussion. Consider an intersection between two roadways of two-way traffic. At the point of intersection each roadway has one or more approach lanes for through traffic in each direction, and an approach lane for traffic turning left through the intersection. Design a system to control the traffic lights at this intersection in such a way as to optimize traffic flow. The system can utilize sensors to sense passing vehicles, and computers to analyze the traffic flow and adjust light timing and sequencing as appropriate.

This problem is quite simple compared to the problems that designers of large software systems must face nowadays; nevertheless it provides abundant illustrations of scenarios in use. It is also an area of current active technological development. The British Government's Road Research Laboratory has recently developed a system called Microprocessor-Optimized Vehicle Actuation (MOVA) that alters light timings to minimize the amount of time vehicles have to wait at intersection [10]. The City of Los Angeles has designed and implemented a system called Automated Traffic Surveillance and Control (ATSAC) that is currently being used to control traffic at over one thousand intersections [20]; by 1998 all 4,000 signalized intersections in the city will be integrated into the system [18]. Computer-controlled traffic signals are increasingly being recognized as having a critical role to play in relieving congestion, and reducing fuel consumption and emissions, and offer a cost-effective alternative to building new freeways and rail lines in urban settings.

Although individual traffic signal control systems have limited complexity, they must interact with a complex environment, namely vehicular traffic flow. The designer of such a system must make sure that the important characteristics of this environment have been taken into account.

3 A Scenario is ...

This section will elaborate upon our high level, initial characterization of scenarios, namely that scenarios are partial descriptions of system and environment behavior arising in restricted situations.

Scenarios are *partial* descriptions because they need not completely specify all the states that comprise a behavior, nor need they completely specify all the attributes of any given state. The intent is to allow the communicator of a scenario to provide only those parts of the description that are relevant to the scenario. For example, it is this characteristic that

allows one to describe a scenario of a car moving through an intersection, without mentioning the number of passengers, or describing the buildings located around the intersection.

Scenario descriptions are formed of two parts — *behavior* and *situation*. These are expressed using the same constructs, but kept separate for purposes of focus, a distinction we will explain shortly. The behavior and the situation are each a partial ordering of states and transitions. Ordering between states and composition of partially ordered states are specified by means of the following common relations [11]: sequentiality (e.g., one state must follow another, or one behavior must follow another), partial and total concurrency (e.g., two events must take place at the same time), repetition, permutation, inclusive and exclusive or (e.g., event1 or event2 but not both), negation (e.g., event3 is not to occur) etc. For example, the behavior of a car moving through an intersection may be specified as a sequence of states (car in front of intersection, car inside intersection, car beyond intersection). Note that the scenario communicator has a variety of options for describing such a behavior — as an alternative to the sequence of *states* just mentioned, he/she could instead provide a sequence of *transitions* (car approaches the intersection, car enters the intersection, car leaves the intersection), or a combination of states and transitions (e.g., car is to the north of the intersection, car enters the intersection, car is to south of the intersection). These alternatives allow the communicator in each case to describe the same partial behavior, but with respect to different attributes of the system and environment. This illustrates how the communicator may describe partial behaviors so as to *focus* on desired portions of system and environment behavior.

Another aspect of *behavior* which is separate from the aforementioned partial description of states and transitions is the *rationale* of the scenario. Rationale is formalized in terms of a completely distinct set of relations among states and transitions. Such relations include triggering, enabling, and restraining. The effect of these is to further restrict the behavior described by the scenario. Again considering any of the previous scenarios about cars moving through an intersection, we know that entering the intersection is not an unconstrained transition. Rather, there are both enabling conditions which make entry into the intersection possible (green lights, ...), as well as restraining conditions which prevent it (red lights, ...). Such conditions on the temporal orderings within scenarios are common.

A scenario's *situation* is specified using the same constructs as its behavior, but is kept separate from that description in order to distinguish *focus*. Typically, the situation is where we would state background assumptions on the initial state of the scenario, and on the progress of the scenario (e.g., invariants that must hold throughout). For example, in the above the scenario communicator has focused on the movement of the car. The situation is the state of the traffic light. The situation expresses how the traffic light changes (red to amber to green to red ...). A change of focus by the scenario's user (e.g., focusing on how the traffic light responds to the presence or absence of cars) will cause behaviors to be reclassified such that the car's activity becomes part of the situation and the traffic light's activity becomes part of the behavior of interest. Although the basic formalism we use is similar to that in other scenario-based research, the separations we make between behavior

and situation, and between rationale and temporal orderings, are relatively novel.

The focus of a scenario is often a consequence of the scenario's intended communicative function. That is, when a scenario is being used to demonstrate some point about system functionality, either the situation or the behavior is being demonstrated. The scenario may be describing the behavior expected in a given situation, or a situation in which a given behavior is expected to occur.

Distinct from the above discussion is the abstraction level at which a scenario is described. It is crucial that scenarios be expressible in concrete terms, yet at arbitrary levels of abstraction. This is accomplished by reifying abstract concepts as concrete instances. For example, suppose a hierarchy of concepts included "moving-vehicle," "car," and "Ford Taurus," then any one of those abstract concepts could be "reified" into an object (or multiple such objects) to be used in scenarios. This capability allows the analyst to describe a behavior at the highest level of generality, rather than at a lower level which might require details immaterial to the intent of the scenario communicator. With regard to our example, the intersection concept has been reified as a generic instance of an intersection which is used instead of some specific intersection. Alternatively, if the attributes of a specific intersection were important to the scenario communicator, then the more specific instance would be used. This illustrates once again the pervasive role *focus* plays in determining what is important and thus what should and should not be included in a scenario.

Although scenarios can incorporate abstractions, it is important that scenarios not be too abstract or general, lest they ignore important factors in the situation. It is not always obvious a priori what factors in a situation have impact on a scenario. For example, one might imagine that it is unnecessary to describe the buildings surrounding an intersection when designing a traffic light controller, but this is not always the case. Driveways into and out of the surrounding buildings can influence traffic flow; so can displays in surrounding store windows. Scenarios are used to make these influences explicit in realistic situations, so that designers can take them into account. We will see numerous examples of this throughout the paper.

4 Three Example Systems

As part of our overall effort at understanding and supporting the use of scenarios, we have developed or are developing three scenario processing capabilities. These have all been integrated with the our knowledge-based system for supporting requirements acquisition, called ARIES (Acquisition of Requirements and Incremental Evolution into Specifications) [13, 14]. This section introduces the systems briefly, so that the reader may have some understanding of how automated tools can make use of scenarios. At the end of the paper, after the uses of scenarios have been surveyed, we will revisit these systems and further categorize their capabilities.

The Requirements Acquisition by Demonstration (RAD) system is intended to be used by non-programming application domain experts to create application domain scenarios in which multiple objects are interacting with their environment [15]. The scenarios are

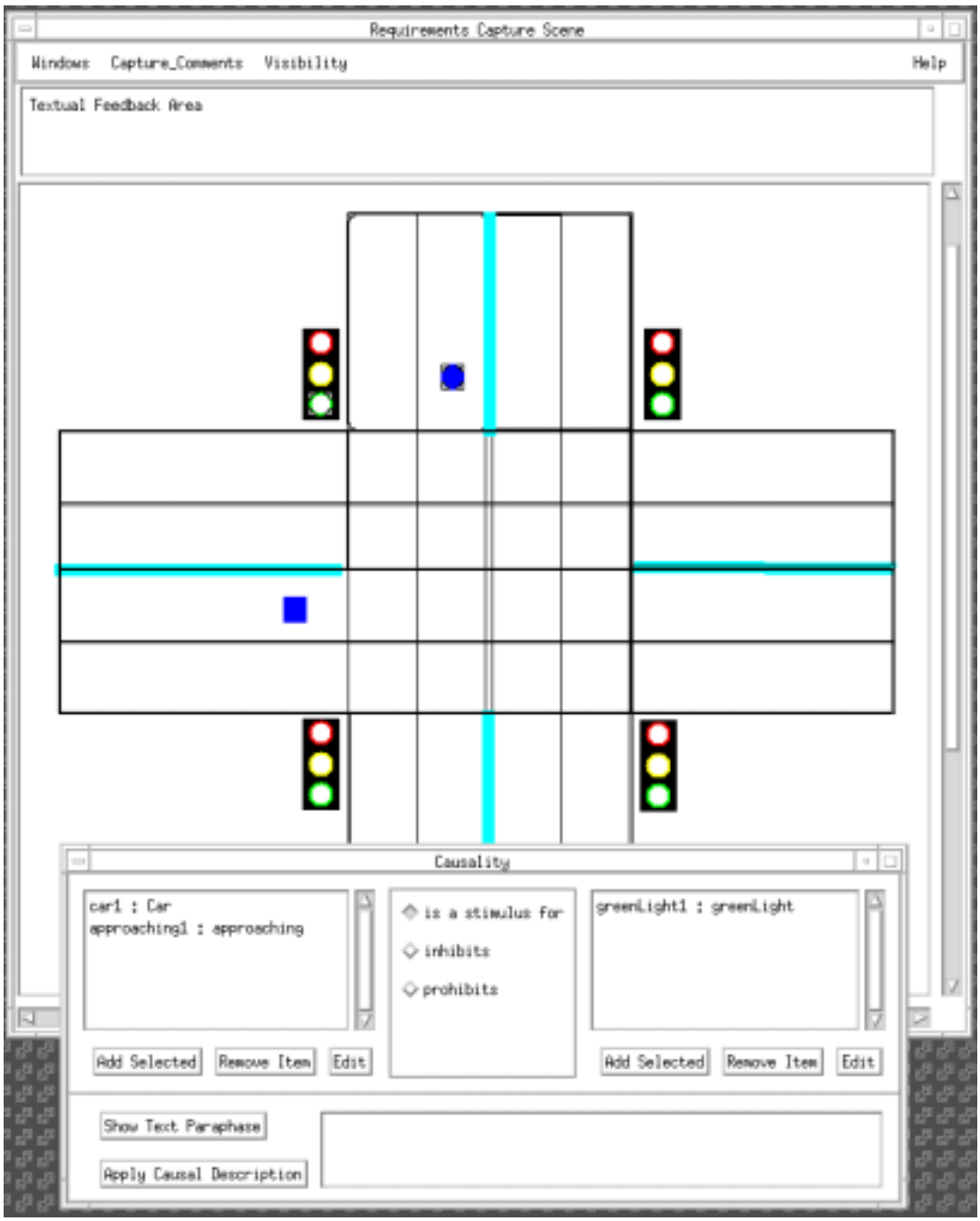

Figure 1: RAD demonstration interface

constructed by manipulating a graphical depiction of a situation into which the system being designed will be introduced, such as the depiction of a traffic light system shown in Figure 1. A user can construct or modify domain specific objects, assign behaviors to individual objects in the situation, and describe the rationale among the objects interacting in the situation. An interactive forms-based dialog is used to describe the behavior so that the scenario can be animated. RAD also allows the domain expert to describe scenarios at different levels of abstraction as well as describe the focus for the scenario being demonstrated.

The ARIES Simulation Component (ASC) addresses validation of specifications to make sure they encompass the right behaviors. In particular, ASC allows an analyst to state validation questions as scenarios which ASC then answers via simulation. The general form of the validation question is: "Does some specific behavior happen in the given situation." The ASC simulator answers this question by determining if the behavior is realizable, sometimes realizable, not realizable, or sometimes not realizable in the current specification. ASC differs from other simulation based tools in that it uses scenarios to address the problems of reducing irrelevant simulation data, simulating incomplete specifications, and reducing the size of the behavior space to consider.

ASC aids an analyst in abstracting out of the specification an appropriate simulation model, called a *specialized specification.* The specialized specification — based on the context of the validation question's behavior and situation — preserves the details of system behaviors that have a bearing on the validation question scenario, and suppresses other details. If the specification is incomplete, the specialized specification is augmented in order to make it executable, as in the PAISLey simulator[21]. This augmentation is incorporated as part of a validation question's situation. Finally, scenarios may be used to capture simplifying assumptions, also as part of a validation question's situation. In general, the specialized specification can be thought of as a slice through the specification's overall behavior space which includes only those behaviors concerned with the validation question scenario.

ARIES has an on-line instruction facility that employs scenarios to train users of the ARIES system. Each scenario describes a sequence of actions that a user might take in performing some task when using the system. The system automatically sets up a situation in which the task is appropriate to perform: it initializes the user's workspace, and displays on the screen those elements of the workspace that the user will manipulate while performing the task. When the user activates a training scenario, the system brings up a window that shows the steps in the scenario, as shown in Figure 2. The display shows a three-level hierarchy of structure in the examples: tasks to be performed (e.g., "Perform a complex evolution"), actions to complete as part of the task (e.g., "Find transformation"), and individual buttons to mouse on or text to type (e.g., "modify-spec"). When the mouse is placed over one of the action descriptions, a detailed description of each input required appears in a documentation window. The left column of the display indicates the analyst's progress through the example:"C" indicates that the action has been completed, and an arrow indicates the action that should be performed next. The instructional system tracks

the user's progress through the scenario, and allows him or her to make detours from it at any time.

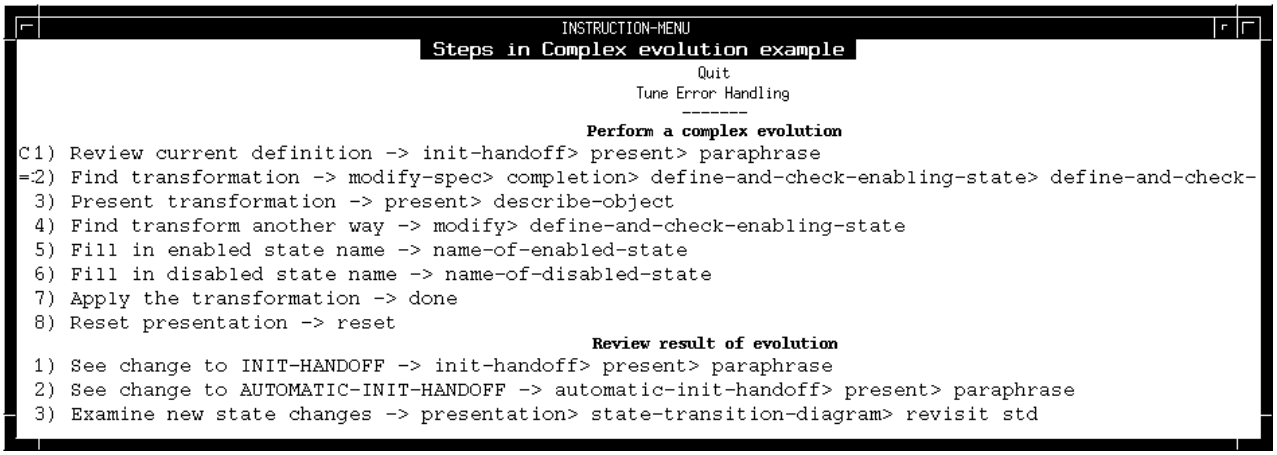

Figure 2: An instructional scenario

5 Roles of scenarios

We use the controller for traffic lights of a 4-way intersection as a common example throughout this section. This is intended as an intuitive example (of a system we are all familiar with) that illustrates the activities and corresponding capabilities we are postulating are useful.

5.1 Describing and clarifying the relevant properties of the application domain

Curtis et. al. [7] cites "the thin spread of application domain knowledge" as an important cause of development effort and mistakes in large software systems. Capturing domain knowledge in the form of scenarios can alleviate this problem. Scenarios are also useful to validate our understanding of domains.

In the context of our 4-way intersection example, examples of domain properties that we would wish to ascertain include vehicular flow and velocity profiles of vehicles approaching, traversing and departing intersections, safety requirements in the form of bounds on amber light times, the lengths of time it takes the traffic in an intersection to clear out (when the lights change), etc. We examine these in more detail in the following subsections.

Vehicular flow around intersections: Vehicle behavior in the vicinity of intersections can be ascertained by considering a variety of typical situations (when approaching a clear intersection with a green light, when in a line of traffic, when approaching a red light, when accelerating on exit from an intersection, etc.).

From this we would determine, for example, lower bounds on the typical distances before and after intersections at which vehicles are flowing at full speed (i.e., have not begun decelerating in approach to the intersection, and are not still accelerating having traversed an intersection).

We would also determine that, in general, the longer the green light time (i.e., the longer each light stays green before turning to red and the next light in the sequence turning to green), the better the average throughput (because each change of lights necessitates the expense of stopping all traffic from entering until traffic within the intersection has cleared out — i.e., "context switching" in our domain). An actual scenario supportive of this is that of the traffic lights that used to be on highway 101 in California (a very busy route — freeway most of its length but interrupted in a few places to allow cross traffic) as it passed through Santa Barbara; anyone who has driven highway 101 through this intersection knows the additional delays incurred by bringing freeway traffic to a halt, and anyone who has waited to use one of the streets crossing highway 101 at this point knows the potentially long wait (several minutes) for a green light (indeed, signs advised drivers to turn off their engines while waiting!).

Bounds on light times: Scenarios can be applied in deriving quantitative safety bounds. For example, consider the task of finding the lower bound on the duration of the amber light, in the transition from a green light to a red light. The following is a relevant scenario:

> Consider the situation of a vehicle approaching an intersection at the time the light changes to amber. The behavior required of the vehicle is that it either comes to a halt before entering the intersection, or enters the intersection prior to the signal changing to red. This can be used to determine a lower bound on the amber period — too short a period would make it impossible for a vehicle approaching at the legal speed limit to either stop (based on driver reaction time and vehicle braking capacity) or have entered the intersection (assuming continued movement at maximum legal speed).

Time for the intersection to clear: In a similar manner to determining bounds on light times, scenarios can be used to determine the time it will take for an intersection to clear of vehicles after a light change:

> Consider the situation of a vehicle that enters an intersection at the last possible instant of an amber light, and compute the time required by the behavior in which the vehicle exits the intersection.

> Consider the situation of several vehicles waiting in the intersection to turn left (as many as can fit in the intersection) and compute the time required by the behavior in which they go from their stationary positions, through their turn, and out of the intersection.

Analytic techniques come into play to perform the actual calculations, but note that a scenario (or several scenarios) provides context within which to do these calculations.

In all these examples, it is the concrete nature of scenarios that identifies what domain properties are relevant, and suggests means to clarify their details.

5.2 Uncovering Requirements

Given a candidate design, the domain expert may draw upon scenarios from the real world to uncover further requirements that might impinge upon the design. Failure to consider these scenarios makes it much more likely that these requirements will be overlooked. In other words, the context provided by real-world scenarios is useful in bringing into consideration issues that lie outside the idealistic boundary we might otherwise have been tempted to draw around a system description.

Placement of loop detectors: Consider the following description on the choice of placement of loop detector (sensors that detect the passage of vehicles) (Figure 3), taken from Rowe [18]:

> "System detectors are placed in either of two configurations: in each marked lane at least 250 feet upstream of the signalized intersection or 100 feet on the departure side of the nearest upstream intersection which is signalized. The second option is more economical because of shorter conduit runs to the nearest intersection controller, but can only be used where there are no significant additions or losses of traffic between the detector and the downstream intersection and where the distances between intersections are not too great."

First, note the quantitative bounds on detector placement: ... *at least 250 feet upstream of the signalized intersection or 100 feet on the departure side of the nearest upstream intersection which is signalized* ... — presumably these bounds were arrived at from an analysis of typical traffic flow to determine where "normal" traffic flow could be measured by sensors (the vehicular-flow example of section 5.1).

Second, note the qualification: ... *but can only be used where there are no significant additions or losses of traffic between the detector and the downstream intersection and* .... This is precisely the sort of requirement that would be uncovered by considering real-world scenarios, e.g., a parking lot between the two intersections that feeds extra traffic into the road downstream of the point where the 100 foot sensor would go. We see the extracted description as the *codification* of a mix of domain knowledge and special-case reasoning.

Figure 3: Placement of system detectors

Downstream blockage / shortcutting: Domain considerations (section 5.1) suggested that the the longer the green light time the better; at this point, therefore, we might expect a domain expert to seek "stress" scenarios illustrating disadvantages of a very long green light. Two such scenarios that the domain expert might come up with are:

> Downstream blockage — a too-long green light lets so much traffic through so quickly that it backs up at the next intersection down the street, nullifying the advantage of the lengthy green light time.
>
> Detouring — a too-long green light, and therefore too-long red light in some other direction(s), leads to people finding alternative routes around the intersection, e.g., residential side streets, thus nullifying the intended use of major thoroughfares.

Notice that both these scenarios uncover pertinent aspects from activities beyond the single intersection that is our primary concern — the first expands consideration to include

the next intersection along the street, while the second deals with how drivers might react to circumvent the behaviors we had planned for them.

5.3 Evaluating Design Alternatives

Scenarios help while investigating alternative designs and requirements. Specific scenarios are especially useful for highlighting the crucial differences between design alternatives; they aid both in establishing what the differences might be (as in the other sections, through consideration of specific cases), and in serving as focal points around which to discuss relative importance (i.e., taking into account the relative frequencies at which scenarios occur, and the relative (dis)advantages of each).

Whether or not to have a left-hand turn arrow? A scenario that demonstrates the utility of a left-hand turn arrow is one in which during the time of a complete light cycle, more than three vehicles arrive along some direction wishing to turn left, and while that direction's light is green, there is continual heavy traffic in the opposing direction (meaning that left turns can be done only on the light changing to red, when the opposing traffic halts, and the three vehicles already in the intersection can complete their left turn). Ultimately, the accumulation of traffic wishing to turn left fills the left hand turn lane and backs up into the straight lane, impeding the flow of all traffic through the intersection.

Applying this same traffic flow behavior to a different situation, that of an intersection with a separate left-hand turn arrow, shows how the left hand turning traffic is allowed through, and does not therefore back up.

Conversely, a scenario in which there is sporadic traffic flow in the opposite direction, or only occasional vehicles wishing to turn left, illustrates how traffic flows satisfactorily even without a left-hand turn arrow, and may even demonstrate better overall throughput for the intersection (because of not having to devote some of the light cycle time aside specifically for left hand turns).

Balancing the considerations that arise in each of these scenarios provides the analyst with the motivation for choice of one design over another.

Whether to coordinate closely spaced signals? We have already observed that traffic blockage can occur when traffic waiting at one signal backs up as far back as another signal. This is especially likely in the case of closely spaced signals. Two alternative designs to dealing with this are to either coordinate the light cycles of those two signals, or to leave those signal cycles uncoordinated, but have short cycle times.

The choice will depend on the situation. The former design — coordinated cycles — is appropriate for scenarios of heavy traffic flow, with an objective of maximizing throughput. For example outside our office building is a lot of commuter traffic flowing along a short cross street that intersects two parallel, heavily traveled streets; since the objective of traffic planners in this situation was (presumably) to maximize throughput, the light cycles at the intersections of the cross street with the two other streets are carefully coordinated, allowing

long green lights (and thus good throughput), but avoiding downstream blockage along the cross street by coordinating those green light times. Conversely, another Los Angeles situation is that of the Wilshire shopping district; here, the roadside stores welcome stop-and-go slowly moving traffic, because it gives people the opportunity to notice the stores they are passing. Thus in this area, short light cycle times avert downstream blockages (it is still an objective that traffic flow, albeit slowly) and produce the kind of traffic flow desired for a shopping district.

5.4 Validating Designs

Analysts and domain experts often use scenarios informally to validate designs throughout the development process. They employ their knowledge of the domain and the current design to develop and reuse individual scenarios which selectively stress the current design while insuring that the requirements are satisfied. This approach differs from exhaustive testing in that it neither requires the design to be complete, nor requires the commitment of resources necessary for exhaustive testing. The following paragraphs demonstrate the use of scenarios which stress important parts of a design.

Amber light duration: Restating the issues relating to the duration of an amber light: If it is too short, cars will end up in the intersection when the light turns red. If it is too long the intersection becomes idle unnecessarily or the amber may be ignored by motorists. To address these problems an analyst has selected a design in which the amber duration is as short as possible, yet sufficiently long to ensure our safety concerns. Given a specification with this bias, an analyst might construct the following validation scenario to stress the safety portion of the requirements:

> Situation: A car is approaching an intersection and the controlling light turns amber. Consider the expected speed of the car and the rate at which the car can decelerate.
>
> Behavior: Does the car have time to either continue through the intersection before the light turns red or come to a stop before entering the intersection?

While the above scenario represents a whole class of behaviors, analysts and domain experts do not exercise all possible instances of this class. Instead one or more representative behaviors are selected. In this case, since the concern is safety, the analyst selects the worst case one, i.e., the highest expected speed and the slowest expected deceleration rate. Confidence that this executes properly translates into confidence that the entire class of scenarios executes properly.

Coordinated lights vs. short lights: Selection of one design over the other in the case of coordinated lights verses short duration lights depends on the performance of these designs under varying, expected traffic conditions. Scenarios allow the analyst to quantify these conditions as alternative situations and then via simulation stress each of the designs with respect to these situations to determine whether the desired behavior occurs.

The above examples illustrate how scenarios serve three roles in design validation. First, scenarios are used to state user expectations (i.e., in a given situation, what behaviors does the analyst expect?). These expectations may either be bad behaviors which should not occur or be good behaviors that should occur. ASC refers to this class of scenario as a Validation Question.

Second, a scenario situation is used to justify focusing execution on specific parts of the behavior space to the exclusion of others. That is, with respect to a validation question, some parts of the specification are more relevant than others. This realization allows the analyst to expend significantly less effort in order to achieve executability as opposed to making the entire specification executable and then having to deal with the resulting myriad of data.

Third, scenarios can be used to approximate behavior via concrete depictions at an appropriate level of detail. This idea is based on the thesis of this paper, that acquisition, formalization, and reasoning (in this case, execution) of scenarios, within well constrained situations, is easier than doing the same for abstract descriptions of behavior. Such approximations allow the analyst to facilitate early executability and hence design validation.

The utility of each of the above validation scenarios is that the analyst is able to validate the design during the development process rather than waiting until the entire design has been completed, thus uncovering latent requirements and design errors and reducing the amount of rework necessary because of errors discovered late in the development process. Guindon in [9] describes this process with respect to mental simulations. ASC [3] provides computer support for this same activity.

6 Computer-Based Support for Scenarios

We have seen many ways that people use scenarios in the previous examples. There are also many ways that computer-based tools can support scenario processing. The following is a summary of the major kinds of computer-based processing of scenarios that we believe are feasible, and a discussion of the contributions of the systems that we have built in each area.

6.1 Recording scenarios

The most fundamental function that computer-based tools can provide is the means to record scenarios; this serves as a prerequisite for any other computer-based support. Argumentative hypermedia tools such as gIBIS [6] can serve this purpose. Such tools use hypertext nodes to record the arguments for or against particular design choices. One form of argument can be whether or not the system supports a particular scenario, described in natural language. Unfortunately, recording scenarios in natural language limits their usefulness, since computer-based tools cannot provide much assistance in processing the scenarios. Tools that can record scenarios formally in some fashion are likely to be more useful.

Our work has focused more on specialized notations that formalize the *situation* and *behavior* in the scenario. In ASC scenarios are expressed as path expressions [11], a notation designed to describe sequences of events. The ARIES representation for states and events [14] may be used to describe scenario situations. In RAD users first create a domain-oriented model (an interactive depiction) of a situation which contains objects in their environment, and then describe the behavior exhibited by the objects in that situation.

6.2 Retrieving scenarios

Once a group of scenarios has been collected and recorded, it is important to be able to retrieve them whenever they are relevant. For example, when a designer is trying to develop an algorithm for selecting the duration for the left-turn signal, he or she should be able to retrieve all scenarios involving left-turning traffic, or which assign particular durations to the left-turn signal.

At the present time we have made no special provision for retrieving scenarios or pieces of scenarios beyond the general mechanisms within ARIES for storing and retrieving requirements information in a knowledge base.

6.3 Executing scenarios

Executing scenarios can uncover missing or inconsistent information in the scenarios, and in the system being designed. By walking through the scenario and comparing it to a domain model one can determine whether the scenario is missing key steps, and whether implicit assumptions are being made that were not recorded in the scenario. Computer-based testing tools can also use scenarios to drive system execution, and check whether or not the system behaves as indicated in the scenario.

ASC can drive the execution of a specification using scenarios. ASC uses scenarios to model the external environment (external relative to the current focus, i.e., validation question). This enables the analyst to force the simulation to react to specific situations, and validate those reactions. RAD is designed to animate the execution of scenarios, so that domain experts can validate and critique examples of behavior. The completed RAD system will be able to check for violations of requirements and domain axioms automatically as the scenario is being executed, and point out the problems to the user.

6.4 Visualizing scenarios

The power of scenarios in highlighting concrete details of system and environment behavior is further enhanced if the scenarios can be visualized in domain terms. For example, representations of vehicles moving through intersections are more perspicuous to most viewers than animated state-transition diagrams of the same scenarios. Computer graphics tools for animating behavior within situations are an important means for supporting this visualization process.

The domain-oriented depictions in RAD are intended to provide this visualization capability. We expect that giving domain experts such domain-oriented depictions to work with will help uncover important domain properties and requirements that might otherwise be overlooked or ambiguous if the scenarios were described using text. For example, depending on the frame of reference, there are three interpretations for the phrase, "The truck in front of the car." The intended frame of reference might be the car or some outside observer. Each interpretation has a unique visual depiction.

6.5 Monitoring scenario execution

Scenario monitoring involves watching the system as it runs, and detecting whether or not behavior patterns described via scenarios occur.

The ASC simulator is designed to monitor validation question scenarios. In general this is a pattern matching activity in which all simulation events are matched against the validation question scenario as they occur. ASC is then able to inform the analyst of matches, partial matches, and contradictions as they occur. The analyst then uses this knowledge to decide whether or not the specification is behaving properly with respect to the current validation question scenario.

The ARIES instructional system also performs scenario monitoring—however, it monitors actions by users rather than actions by the system. Each scenario's behavior is encoded as a sequence of user inputs, where each user input may be a keystroke, a sequence of keystrokes, or some class of keystrokes. The user's keystrokes are matched against the scenario description to determine adherence to the training scenario.

6.6 Developing requirements and designs from scenarios

We envision that automated tools will ultimately be able to generate requirements and designs directly from scenarios. It will then be unnecessary to develop scenarios and designs separately and compare one against the other. Some systems that demonstrate such capabilities already exist. For example, the WATSON system can take as input a scenario describing a proposed feature for a telephone system, and derive from it a complete specification of the feature [17].

We are providing some capabilities along these lines in RAD. The system provides a set of generic building blocks to let domain experts express the domain specific concepts that represent the objects and their situation as well as the behavior of the objects in the scenario. They will be able to express the triggering and restraining conditions in the scenarios that must be satisfied at particular states in the scenario. In this way the scenario can be gradually transformed into a rule specifying a particular class of system behavior. Further work would be required to recognize and resolve conflicts between such rules, combine specific rules and generalize them into more complete specifications of behavior.

6.7 Processing scenarios effectively

The extent to which such automated support for scenario processing is effective depends heavily upon being able to refer to some existing body of formalized domain knowledge. Without such domain knowledge, scenarios are cumbersome to state, and automated reasoning on scenarios is likely to yield nonsensical conclusions. Suppose, for example, that in writing the scenarios described in the previous section, it was necessary to define each time what a vehicle was, what a roadway was, what traffic flow was, etc. Recording the scenarios would quickly have become a very burdensome process. The RAD system provides a set of useful scenario building blocks that the user can employ and further compose, easing the task of capturing scenario descriptions. Retrieval of scenarios presupposes some sort of content-oriented classification of scenarios; a knowledge base of terminology for classification would be needed for this purpose.

If a design has already been developed, it may be possible to write scenarios in terms of the concepts already identified and defined in the design description. This does not help for scenarios that are being articulated prior to design, however. In such cases it is necessary to rely upon some preexisting domain model or knowledge base describing the domain.

7 Summary

This paper has demonstrated the importance of scenarios in the software engineering process. They are useful throughout the process, not just in specific activities such as testing. They are important not just for systems with human interfaces, but for any system that interacts with a complex environment, and most likely for complex systems in general. We believe that they have been neglected in part because tools that support the recording and processing of scenarios as first-class artifacts have been lacking. This in turn may be a consequence of a tendency within computer science to emphasize the general over the specific, and the abstract over the concrete.

Tools are being developed by us as well as others that support scenario processing. They are beginning to provide enough in the way of processing support to make them worth using. Evaluation studies with domain experts and software engineers need to be conducted in order prove their effectiveness.

References

[1] J.R. Anderson. Acquisition of Cognitive Skill. *Psychological Review*, 89(4), 1982.

[2] J.S. Anderson and B. Durney. Using Scenarios in Deficiency-Driven Requirements Engineering. In *RE'93: IEEE International Symposium on Requirements Engineering*, pages 134–141. IEEE Computer Society Press, January 1993.

[3] K.M. Benner. Specification Reformulation During Specification Validation. In *Proceedings, Workshop on Problem Reformulation and Representation Change, Asilomar Conference Center, Pacific Grove, CA*, April 1992.

[4] R.L. Campbell. Will the Real Scenario Please Stand Up? *SIGCHI Bulletin*, 24(2):6–8, April 1992.

[5] J.M. Carroll and M.B. Rosson. Getting Around the Task-Artifact Cycle: How to Make Claims and Design by Scenario. *ACM Transactions on Information Systems*, 10(2):181–212, April 1992.

[6] J. Conklin and M.L. Begeman. gIBIS—A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331, 1988.

[7] B. Curtis, H. Krasner, and N. Iscoe. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, 31(11):1268–1287, November 1988.

[8] S.F. Fickas and P. Nagarajan. Critiquing Software Specifications: A Knowledge Based Approach. *IEEE Software*, 5(6):37–49, November 1988.

[9] R. Guindon. Knowledge Exploited by Experts During Software System Design. *Int. J. Man-Machine Studies*, 33:279–304, 1990.

[10] M. Hamer. Traffic Lights Learn To Go With the Flow. *New Scientist*, 1715:35, May 1990.

[11] W. Hseush and G.E. Kaiser. Modeling Concurrency in Parallel Debugging. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 11–20. ACM Press, March 1990. Published in *SIGPLAN Notices*, Vol. 25 No. 3, March 1990.

[12] W.L. Johnson. Specification via Scenarios and Views. In *Proceedings of the 3rd International Software Process Workshop*, pages 61–63, Breckenridge, CO, 1986. IEEE Computer Society Press.

[13] W.L. Johnson, M.S. Feather, and D.R. Harris. Integrating Domain Knowledge, Requirements and Specifications. *Journal of Systems Integration*, 1:283–320, 1991.

[14] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and Presentation of Requirements Knowledge. *IEEE Trans. on Software Engineering*, 18(10):853–869, October 1992.

[15] W.L. Johnson and L. Zorman. Supporting Groups Involved in Software Specification and Design. In *Proceedings of the 1992 CSCW Workshop on Understanding and Supporting Successful Group Work in Software Design*, 1992. Available from John Karat, IBM Watson Research Center.

[16] J. Karat and J.L. Bennett. Using Scenarios in Design Meetings - A Case Study Example. In *Taking Software Design Serious Seriously: Practical Techniques for Human-Computer Interaction Design*, pages 63–94. Academic Press, 1991.

[17] V.E. Kelly and U. Nonnenmann. Reducing the Complexity of Formal Specification Acquisition. In *Proceedings of the AAAI-88 Workshop on Automating Software Design*, pages 66–72, St. Paul, MN, 1988.

[18] E. Rowe. The Los Angeles Automated Traffic Surveillance and Control (ATSAC) System, September 1992. Los Angeles Department of Transportation.

[19] L.A. Suchman. *Plans and Situated Actions: The Problem of Human-Machine Communication.* Cambridge University Press, New York, 1987.

[20] J. Walters. Los Angeles: Technology and Traffic. *Governing*, page 43, October 1992.

[21] P. Zave. An Insider's Evaluation of PAISLey. *IEEE Trans. on Software Engineering*, SE-17:212–225, 1991.